- Mark your completed exercises in the OLAT course of the PS.

- You can start from `template_10.tgz` provided on the proseminar page.

- Your `*.hs` files must be compilable with `ghc`.

- Upload your `*.hs` files in OLAT. Please upload each file separately and do *not* use zip or upload any build files (`*.exe`, `*.o`, `*.hi`, etc.)

Exercise 1 *I/O*   **4 p.**

1. Implement a stand-alone Haskell program that prints the name of all .txt files in the current directory prefixed by a number, starting from 1. For instance, if the current directory contains files

   ```
   Main.hs
   foo.txt
   whatever.jpg
   bar.txt
   ```

   then starting the program from the command line should print the following two lines:

   ```
   1: foo.txt
   2: bar.txt
   ```

   Try to separate the I/O task and formatting the list of file names into different functions. You might find it helpful to define auxiliary functions `filterTxtfiles :: [FilePath] -> [FilePath]` and `formatFiles :: [FilePath] -> String`. (1.5 points)

   Hint: A Haskell function that returns a list of files in a given directory can be found in the `System.Directory` documentation.[1] The function `isSuffixOf`[2] might be useful to check whether a file ends in `".txt"`.

2. Add interaction to your program. First, print the .txt files in the current directory, as in the previous task, then read a number input by the user. If the user inputs 0, the program should quit, otherwise the contents of the .txt file with the given index should be printed.

   You may assume that the user always gives a valid input and that the functionality from exercise 1.1 is available. An example program run could look like this:

   ```
   ... program starts ...
   1: foo.txt
   2: bar.txt
   Input 0 to quit or a file number to view the file contents:
   1
   ... print the contents of foo.txt ...
   ```

   (2 points)

3. Extend your program so that after a file has been printed, the user is asked to proceed by pressing the enter key. After pressing enter, the program should start again. You may assume that the functionality from question 1.2 is available even if you did not solve this exercise.

---

[1] https://hackage.haskell.org/package/directory-1.3.8.1/docs/System-Directory.html
[2] https://hackage.haskell.org/package/base-4.19.0.0/docs/Data-List.html#v:isSuffixOf

```
... program starts ...
1: foo.txt
2: bar.txt
Input 0 to quit or a file number to view the file contents:
1
... print the contents of foo.txt ...
Press enter to restart
... user presses enter ...
1: foo.txt
2: bar.txt
Input 0 to quit or a file number to view the file contents:
0
... program quits ...
```

(0.5 points)

Exercise 2 *Connect Four*   **6 p.**

In this exercise, you will extend the implementation of Connect Four from the lecture in various ways. Note that all subtasks can be solved independently of one another.

1. The user interface does not check whether input moves are valid: it is not checked whether the input from the user really is a number or whether an input number is a valid move. Both cases may lead to unintended behavior or crashes of the program. Therefore, you should modify the user interface so that it repeatedly asks for input until a valid move has been entered, e.g. as follows:

```
Choose one of [0,1,2,3,4,5,6]: five
five is not a valid move, try again: 8
8 is not a valid move, try again: 3
... accept and continue ...
```

(2 points)

2. Extend the Connect Four implementation so that it can save and load games, e.g. in a file `connect4.txt`. The user interface might look like this:

```
Welcome to Connect Four
(n)ew game or (l)oad game: l
... game starts by loading state from connect4.txt ...
Choose one of [0,2,3,5,6] or (s)ave: s
... game is saved in file connect4.txt and program quits ...
```

For the implementation, note that `read . show = id` and that one can automatically derive `Read` instances in datatype definitions via `deriving Read`. (2 points)

3. Modify the function `winningPlayer` in the game logic so that diagonals are also taken into account. (2 points)